Billion-Particle SIMD-Friendly Two-Point Correlation on Large-Scale HPC Cluster Systems

Jatin Chhugani†, Changkyu Kim†, Hemant Shukla⋆, Jongsoo Park†,
Pradeep Dubey†, John Shalf⋆ and Horst D. Simon⋆
†Parallel Computing Lab, Intel Corporation ⋆Lawrence Berkeley National Laboratory

***Abstract*—**

Two-point Correlation Function (TPCF) is widely used in astronomy to characterize the distribution of matter/energy in the Universe, and help derive the physics that can trace back to the creation of the universe. However, it is prohibitively slow for current sized datasets, and would continue to be a critical bottleneck with the trend of increasing dataset sizes to billions of particles and more, which makes TPCF a compelling benchmark application for future exa-scale architectures.

State-of-the-art TPCF implementations do not map well to the underlying SIMD hardware, and also suffer from load-imbalance for large core counts. In this paper, we present a novel SIMD-friendly histogram update algorithm that exploits the spatial locality of histogram updates to achieve near-linear SIMD scaling. We also present a load-balancing scheme that combines domain-specific initial static division of work and dynamic task migration across nodes to effectively balance computation across nodes.

Using Zin supercomputer at Lawrence Livermore National Laboratory (25,600 cores of Intel® Xeon® E5-2670, each with 256-bit SIMD), we achieve 90% parallel efficiency and 96% SIMD efficiency, and perform TPCF computation on a 1.7 billion particle dataset in 5.3 hours (at least 35× faster than previous approaches). In terms of cost per performance (measured in flops/$), we achieve at least an order-of-magnitude (11.1×) higher flops/$ as compared to the best known results [1]. Consequently, we now have line-of-sight to achieving the processing power for correlation computation to process billion+ particles telescopic data.

I. Introduction

Correlation analysis is a widely used tool in a variety of fields, including genetics, geology, etc. In the field of astronomy, Two-Point Correlation Function (TPCF) helps derive the parameters and the underlying physics from the observed clustering patterns in large galaxy surveys that can *trace back to the conditions at the very beginning of the creation of the Universe*. The current sizes of observed datasets are in the order of *hundred of millions of particles* and are rapidly increasing and expected to cross tens of billions of particles soon [2], [3]. However, processing this data to compute the correlation analysis is lagging far behind. For example, TPCF computation on a 1 billion particle dataset would typically consume more than a few exaflops of computation, and take *more than 50 years* on a single-threaded scalar code. Hence, computing TPCF in a reasonable amount of time on massive datasets requires large scale HPC machines, and with the trend of growing sizes, is a tailor-made application for exa-scale machines.

One of the *biggest challenges* in mapping TPCF to large node clusters is *load-balancing across the tens of thousands of cores* spread across thousands of nodes. TPCF implementation typically involves constructing acceleration data structures like kd-trees, and traversing the resultant trees to compute correlation amongst particles of the nodes. The computation involved for these nodes varies dramatically across tree-nodes, and cannot be predicted apriori. Thus, it is challenging to evenly distribute work across cores, and implementations typically under-utilize the computational resources [1], even for as low as 1K cores. Hence there is a *need to develop load-balanced algorithms for TPCF computation* to fully utilize the underlying resources.

Another challenge for large-scale cluster computation is the total expenditure, which is being dominated by the electricity costs related to energy consumption [4], [5]. This is projected to get even more expensive with the increasing number of nodes in the future. Thus, chip manufacturers have been increasing single-node compute density by increasing the number of cores, and more noticeably, the SIMD width. SIMD computation is an energy efficient way of increasing single-node computation, and has in-fact been getting wider – from 128-bit in SSE architectures, 256-bit in AVX [6] to 512-bit in the Intel® Xeon Phi™ [7], [8]. Hence, it is crucial to design algorithms that can fully utilize the SIMD execution units. A large fraction of TPCF run-time is spent in performing histogram updates, which do not map well to SIMD architectures [9], and current implementations achieve negligible SIMD scaling. Thus, *lack of SIMD scaling poses a serious challenge to scaling TPCF* on current and upcoming computing platforms.

Contributions: We present the **first SIMD-, thread- and node-level parallel friendly algorithm for scaling TPCF** computation on datasets with billions of particles on Petaflop+ high performance computing cluster systems with tens of thousands of SIMD cores. Our key contributions are:

- **Novel SIMD-friendly histogram update algorithm** that exploits the spatial locality in updating histogram bins to achieve near-linear SIMD speedup.
- **Load-Balanced computation** comprising an initial static division of work; followed by low-overhead dynamic migration of work across nodes to scale TPCF to tens

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

of thousands of SIMD cores.

- **Communication overhead reduction** by overlapping computation with communication to reduce impact on run-time and improve scaling. Additionally, we also **reduce the amount of data communicated** between nodes, thereby reducing the energy consumption, and increasing the energy efficiency of the implementation.
- Performing TPCF computation on more than billion particles on a large-scale Petaflop+ HPC cluster in a reasonable amount of time of a few hours (details below). To the best of our knowledge, this is the **first paper** that reports TPCF performance on this scale of dataset and hardware.

We evaluate our performance on a 1600-node Intel Xeon E5-2670 Sandybridge cluster, with a total of 25,600 cores each with a peak compute of 41.6 GFLOPS/core for a total peak compute of 1.06 PFLOPS. On our largest dataset with 1.7 X 10^9 particles, we achieve a SIMD scaling of 7.7× (max. 8×), a thread-level scaling of 14.9× (max. 16×) and a node-level scaling of 1550× (max. 1600×) to achieve a performance efficiency of 86.8%. Our SIMD friendly algorithm and the dynamic load-balancing scheme coupled with communication/computation overlap help improve the performance by a minimum of 35× as compared to previous approaches. This reduction in run-time directly translates to a corresponding energy reduction, which is further boosted by the reduction in inter-node communication.

On our input dataset with 1.7 billion particles, TPCF computation takes *5.3 hours*[1] on the above cluster. Coupling our algorithmic novelty with an implementation that maximally exploits all micro-architectural dimensions of modern multicore/manycore processors, we have achieved significantly more than an order-of-magnitude efficiency improvement for TPCF computation. This has the potential for bringing an unprecedented level of interactivity for researchers in this field.

II. Background

A. Motivation

The $\mathcal{N}$-point correlation statistics is used in a wide range of domain sciences. In the field of astrophysics and cosmology, the correlation statistics is a powerful tool and has been studied in depth. For example, correlation statistics is used to study the growth of initial density fluctuation into structures of galaxies.

In the recent decades, various experiments have validated that of the total matter-density of the universe, only 4% comprises everything that we observe as galaxies, stars, and planets. The remainder of the universe is made of something termed as dark energy (73%) and dark matter (23%) [10]. Dark energy is a hypothetical form of energy that permeates all of space and tends to accelerate the expansion of the universe [11]. The work that led to this discovery of accelerating expansion of the universe was awarded the Nobel Prize in Physics in 2011 [12]. Despite various and independent confirmations of the estimates, the nature of these physical components are largely unknown.

Consequently, there are multiple ongoing efforts worldwide to conduct very large (sky coverage) and deep (faint objects) sky surveys. The large surveys are used to glean various cosmological parameters to constrain the models. A class of cross and auto-correlation functions helps derive parameters and the underlying physics from the observed clustering patterns in large galaxy surveys. Fourier transform of correlation functions yields the power spectrum of the matter density field. The decomposition of the field into Fourier components help in identifying physical processes that contributed in the density perturbations at different scales, unfolding the interplay of dark matter and matter in the early universe and the effects of acceleration due to dark energy relatively later. Studying the large-scale distribution (clustering due to gravity) patterns of matter density at various stages of cosmic evolution sheds light on the underlying principles that can trace back to the conditions at the very beginning of the creation of the universe.

The size of the datasets captured has been growing at a rapid pace. From relatively small datasets with around 10M galaxies in 2011, the datasets have already crossed a Billion galaxies and expected to cross tens of billions of galaxies by the end of the next year [2]. This represents an increase of 3 orders-of-magnitude in less than 3 years. The recently announced Square Kilometer Array (SKA) radio telescope project is expected to produce 2 times the current internet traffic rates (1 exabyte) by the next decade [13]. The LSST data challenge [14] and the IBM 'Big Data Challenge' [15] are some examples that further strengthen the need to develop efficient high-performance tools. In fact, correlation computation (and its variants) has been identified as one of the Top-3 problems for the data generated by next generation telescopes [16].

Current TPCF computation rates are too slow and hence impact the size of datasets that scientists typically run on. For example, a 1.7 billion dataset using current algorithms would take more than a month [2] on a Petaflop cluster, and is clearly infeasible for analyzing results on real/simulated data. Hence, it is imperative to develop algorithms that speedup the processing times for such datasets to a more tractable range (*around a few hours*) and scale with future processor trends (larger number of nodes, cores and increasing SIMD widths) to be able to efficiently process the increasing dataset sizes and to put us on the path of discovering answers to the ultimate question – 'How was the universe formed and how will it end?'

B. Problem Definition

Notation: We use the following notation throughout the paper:

$\mathcal{D}$: Set of input data particles (point-masses) in 3D space.
$\mathcal{R}$: Set of random data particles (point-masses) in 3D space.
$n_{\mathcal{R}}$: Number of random sets generated.
$\mathcal{N}$: Number of input data particles.

[1] the number of random sets generated ($n_{\mathcal{R}}$) equals the typically used value of 100.

[2] projected using performance numbers by Dolence and Brunner [1]

$\mathcal{D}im$: Dimension of the box (cube) that contains all the particles.
$\mathcal{H}$: Number of histogram bins.
r_{min} : Minimum distance (expressed as a % of $\mathcal{D}im$) that is binned.
r_{max} : Maximum distance (expressed as a % of $\mathcal{D}im$) that is binned.
$\mathcal{DD}(r)$: Histogram of distances between the input particles.
$\mathcal{RR}_j(r)$: Histogram of distances within $\mathcal{R}_j$ ($j \in [1 \,..\, n_{\mathcal{R}}]$)
$\mathcal{DR}_j(r)$: Histogram of distances between $\mathcal{D}$ and $\mathcal{R}_j$ ($j \in [1 \,..\, n_{\mathcal{R}}]$)
$\omega(r)$: Two Point Correlation Function (defined in Eqn. 1).

Although there exist different estimators for the two point correlation function (TPCF) [17], [18], they are all histogrammed functions of the distances, denoted by $\mathcal{DD}(r)$, $\mathcal{RR}_j(r)$, and $\mathcal{DR}_j(r)$ (defined above). Similar to other works [19], [20], [1], we use the Landy-Szalay estimator [18], $\omega(r)$, which is defined as:

$$\omega(r) = 1 + \frac{n_{\mathcal{R}} \cdot \mathcal{DD}(r) - 2\sum_{j=1}^{n_{\mathcal{R}}} \mathcal{DR}_j(r)}{\sum_{j=1}^{n_{\mathcal{R}}} \mathcal{RR}_j(r)} \quad (1)$$

The distance between two input points P and Q (denoted by *dist*(P, Q)) is the euclidean distance between the points. Since the clustering patterns are of importance across a wide range of distance scales [18], [19], the binning schema used in the field of astronomy is typically *logarithmic*. Hence, the bins are *equally* spaced in the logarithmic space, and therefore the bin boundaries (of the $\mathcal{H}$ bins) form a *geometric progression*.

The complete algorithm proceeds as follows: Given a set of $\mathcal{N}$ input particles ($\mathcal{D}$), $n_{\mathcal{R}}$ instances of random particles ($\mathcal{R}_j$, $j \in [1 \,..\, n_{\mathcal{R}}]$), each with $\mathcal{N}$ particles, are created [18], [20]. To compute $\mathcal{DD}(r)$, for each unique pair of particles in $\mathcal{D}$, the distance (r) is computed, and if $r \in [r_{min} \,..\, r_{max}]$, the appropriate bin computed, and $\mathcal{DD}$ histogram's bin incremented. $\mathcal{RR}_j(r)$ is computed similarly. To compute $\mathcal{DR}_j(r)$, the pairs of particles are formed from particles belonging to $\mathcal{D}$ and $\mathcal{R}_j$, respectively. In addition to computing the TPCF, the error estimates can be computed, using jackknife re-sampling [21]. Since this is computationally similar to computing the TPCF [19], we focus on computation of TPCF (Eqn. 1).

To compute each of $\mathcal{DD}(r)$, $\mathcal{RR}_j(r)$, and $\mathcal{DR}_j(r)$, a naïve implementation would consider every possible pair of particles, resulting in a time complexity of O($\mathcal{N}^2$). To *accelerate the computation*, a dual-tree data structure is often used [22], [23], which is described below.

C. Dual-Tree Algorithm

Dual-trees refer to kd-trees [24] built individually around the input and random datasets. Dual-trees accelerate TPCF computation by pruning out regions that have distance greater than r_{max} or less than r_{min}. In addition, optimizations are carried out for cases where distances between pair of nodes or pair of point-nodes fall within the same bin, wherein the all-to-all pair computation is avoided, as described later.

```
Dual_Tree_Traversal_and_Histogram_Update() {
   (r_min^n1,n2, r_max^n1,n2) = Compute_Min_Max_Dist(n1, n2);
   if ( (r_min > r_max^n1,n2) || (r_min^n1,n2 > r_max)   return;
   (min_bin, max_bin) = Compute_Min_Max_Bin_ID(r_min^n1,n2 , r_max^n1,n2 );
   if (min_bin == max_bin) { Hist[min_bin] += |n1| *|n2|;  return; }
   else
      if (!leaf(n1) && !(leaf(n2)))  // Case 1: Call
Dual_Tree_Traversal_and_Histogram_Update() with four cross pairs -- (n1.left, n2.left),
(n1.right, n2.left), (n1.left, n2.right), (n1.right, n2.right)
      ...
      else if (leaf(n1) && (!leaf(n2))
       // Case 2: Call Dual_Tree_Traversal_and_Histogram_Update()
       // with two cross pairs -- (n1, n2.left), (n1, n2.right)
      ...
      else if (!leaf(n1) && (leaf(n2))
       // Case 3: Call Dual_Tree_Traversal_and_Histogram_Update()
       // with two cross pairs -- (n1.left, n2), (n1.right, n2)
      ...
      else  //Case 4
         for (n1_i=0; n1_i<|n1|; n1_i++)  {
            (r_min^n1_i,n2, r_max^n1_i,n2 ) = Compute_Min_Max_Dist (n1_i, n2,);
            (min_bin, max_bin)   = Compute_Min_Max_Bin_ID (r_min^n1_i,n2 , r_max^n1_i,n2 );
            if (min_bin == max_bin) { Hist[min_bin] += |n2|; }
            else Compute_Dist_Update_Histogram (n1_i, n2, Hist, min_bin, max_bin);
         }
}
```

Fig. 1: Code-snippet for traversing dual-trees and updating histogram bins. This function, the major part of the baseline dual-tree based TPCF algorithm, is followed after building kd-tree for the input ($\mathcal{D}$) and the random ($\mathcal{R}_j$) datasets.

A high-level pseudo code is given in Fig. 1. A kd-tree for each of $\mathcal{D}$ and $\mathcal{R}_j$'s is constructed (referred to as $\mathcal{T}_{\mathcal{D}}$ and $\mathcal{T}_{\mathcal{R}_j}$ respectively). Consider computation of $\mathcal{DR}_j(r)$. The two trees ($\mathcal{T}_{\mathcal{D}}$ and $\mathcal{T}_{\mathcal{R}_j}$) are traversed from root, and for each pair of encountered nodes (n_1 and n_2), `Compute_Min_Max_Dist` is executed to compute the minimum ($r_{min}^{n_1,n_2}$) and maximum distance ($r_{max}^{n_1,n_2}$) between the nodes.

If ($r_{min}^{n_1,n_2} > r_{max}$) or ($r_{max}^{n_1,n_2} < r_{min}$), then none of the pairs of particles from n_1 and n_2 contribute to the histogram and there is no further need to traverse down the trees. Otherwise, this is followed by executing `Compute_Min_Max_Bin_ID`, that computes the range of histogram bins that overlap with $[r_{min}^{n_1,n_2} \,..\, r_{max}^{n_1,n_2}]$. The resultant minimum and maximum histogram bins are referred to as `min_bin` and `max_bin` respectively. If `min_bin` equals `max_bin` (i.e., all the pairs fall in the same bin), the corresponding histogram index (`min_bin`) is incremented by the product of respective number of particles in the two sub-trees, and *no further traversal* is required. Else, one of the following *four* cases is executed:
Case 1: Both n_1 and n_2 are non-leaf nodes. The four cross pairs of nodes formed by the immediate children of n_1 and n_2 are recursively traversed down.
Case 2: n_1 is a leaf node and n_2 is a non-leaf node. The two cross pairs of nodes formed by n_1 and immediate children of n_2 respectively are traversed down.
Case 3: n_2 is a leaf node, and n_1 is a non-leaf node. Symmetric to case 2.
Case 4: Both n_1 and n_2 are leaf nodes. For each particle in n_1 (n_{1i}), the minimum distance ($r_{min}^{n_{1i},n_2}$) and maximum distance ($r_{max}^{n_{1i},n_2}$) are computed. If these *fall within one bin*, the corresponding histogram index is incremented by the number of particles in n_2. Else, the actual distances between n_{1i} and each

particle in n_2 are computed, and appropriate histogram bins incremented (`Compute_Dist_Update_Histogram`). This process is carried out for all particles in n_1.

$\mathcal{DD}(r)$ and $\mathcal{RR}_j(r)$ are also computed in a similar fashion (using the appropriate trees). Typically, these trees are built with axis-aligned bounding boxes [23].

III. Challenges

As far as performance of TPCF on modern SIMD multi-core processors in a multi-node cluster setting is concerned, the following *two* factors need to be efficiently optimized for:

A. Exploiting SIMD Computation

The SIMD widths of modern processors have been continuously growing. The SSE architecture has a 128-bit wide SIMD, that could operate simultaneously on 4 floating point values, while current AVX-based [6] processors have doubled it to 256-bits. Intel's Xeon Phi processors [8] are further increasing it by 2× to 512-bits. GPUs have a logical 1024-bit SIMD with physical SIMD widths of 256-bits on the NVIDIA GTX 200 series, increasing to 512-bits on the Fermi architecture [25]. Hence it is critical to scale with SIMD in-order to fully utilize the underlying computing resources.

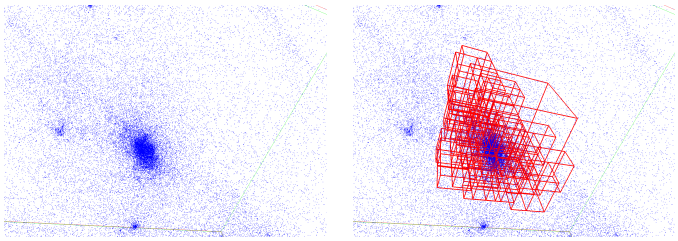
TPCF computation consists of two basic computational kernels – computing distance between points and updating the histogram. A large fraction of the run-time is spent in performing histogram updates, which do not map well to SIMD architecture [9]. This is primarily due to the possibility of multiple lanes within the same SIMD register mapping to the same histogram bin, which requires atomic SIMD updates – something that does not exist on current computing platforms. Most of the recent work on exploiting SIMD on CPUs for histogram updates [26], [27] has resulted in a very small speedup (less than 1.1–1.2×).

In the case of correlation computation (TPCF), we exploit the *spatial locality in updating of histogram bins* to devise a novel SIMD friendly algorithm for *efficiently* computing bin_ids and performing histogram updates. The techniques developed here are applicable to other histogram-based applications with similar access patterns (details in Sec. IV-A). We are not aware of any previous work in this area.

B. Scaling to tens-of-thousands of cores across thousands of nodes

TPCF computation involves traversal of kd-trees and for a given node, computing the neighboring nodes, and the correlation between particles in the corresponding pair of nodes. As shown in Fig. 1, the work done per pair is a function of the distance between the nodes, product of the number of particles in each node, etc. Furthermore, this work can vary drastically across the nodes of the tree.

Such tree-traversal based algorithms pose a challenge for scaling due the irregular amount of work done which cannot be predicted apriori (See Fig. 2). As far as cores on a single-node are concerned, there exist systems [28], [29] which employ a dynamic task queueing model [30] based on task stealing for load-balancing across them. Due to the shared on-chip resources, the resultant overhead of task stealing is less than a few thousand cycles [31], [32], and adds very little overhead, and scales well in practice.



(a) Zoom-in of the input dataset (b) Kd-Tree visualization

Fig. 2: (a) Zoom-in to a region of the input dataset. (b) Visualization of the kd-tree built around the input particles. Only a few tree-nodes are shown for clarity. Note the varying size of the nodes – which captures the non-uniform density of particles in that neighborhood, and prevalent throughout realistic datasets, which poses a challenge for load-balanced computation across tens of thousands of cores.

However, there are relatively fewer systems [33], [34] for scaling such tree-traversal code across thousands of nodes – something we focus on in this paper. This is because the task stealing now occurs across shared-nothing computational nodes, and involves large node-to-node latency and complicated protocols to guarantee termination. The resultant task stealing overheads are now orders of magnitude larger and need to be carefully optimized for.

We perform the work division between the nodes using a two-phase algorithm: a *low-overhead domain specific initial static subdivision of work across the nodes*; followed by *dynamic scheduling involving task stealing (or migration) across nodes* such that all the computational resources are kept busy during the complete execution (Sec. V). This helps us achieve scaling both within the cores of a node, and across nodes in the cluster. In addition, the time spent in data communication also adds up to the total execution time, and can potentially reduce scaling. We propose a hybrid scheme that overlaps computation with communication to reduce the overhead of data communication to obtain near-linear speedups with as large as 25,600 cores across 1,600 nodes.

IV. Our Approach

A. SIMD-Friendly Histogram Computation

Let $|n_i|$ denote the number of particles in node n_i and $\mathcal{K}$ denote the SIMD width (8-wide AVX [6] for CPU) of the architecture. For each pair of kd-tree leaf nodes (n_1 and n_2 respectively), if $[r_{min}^{n_1,n_2} \;..\; r_{max}^{n_1,n_2}]$ overlaps $[r_{min} \;..\; r_{max}]$ in more than *one* bin, then for each particle in n_1 whose extent with n_2 overlaps with more than one bin, we compute its distance with all particles in n_2 and update the relevant histogram bins. Let n_1' denote the set of particles in n_1 whose extent with n_2 overlaps with more than one bin.

Consider the first part of the computation — distance computation. One way of SIMDfying is to perform computation

for each particle in n_1' simultaneously with $\mathcal{K}$ particles in n_2, until all particles have been considered. However, this would require gather operations to pack the X, Y, and Z coordinates of the particles in n_2 into consecutive locations in the registers. To avoid the repeated gather operations, we store particles in a Structure-Of-Arrays (**SOA**) [35]: we store the X coordinates of all particles contiguously, followed by the Y coordinates, and so on. This allows for simply loading X, Y, and Z coordinates of $\mathcal{K}$ particles into three registers before the distance computation. To avoid the expensive square-root operation, we actually compute the square of the distance, and appropriately compare with the square of the bin boundaries. As the next step, we need to update the histogram.

As described in Sec. II-B, bin boundaries are spaced uniformly in the logarithmic space, thereby forming a geometric progression. Let d_0, $d_0\gamma$, ..., $d_0\gamma^{\mathcal{H}}$ denote the bin boundaries ($d_0 = r_{min}$ and $d_0\gamma^{\mathcal{H}} = r_{max}$) . The fraction of volume covered by i^{th} bin ($i \in [1\ ..\ \mathcal{H}]$) is $\sim (\gamma^3 - 1)/\gamma^{3(\mathcal{H}-i+1)}$. Consider a typical scenario with r_{max} = 10% divided into $\mathcal{H}$ = 10 bins. The last four bins cover 99.6% of the total volume (74.9%, 18.8%, 4.7%, and 1.2%, starting from the last bin). After pruning cases where all pairs between kd-tree nodes fall within a bin (since they do not incur any computation), a large fraction of the remaining time is expected to be spent in cases where the range of bins for n_1' and n_2 overlap with a **small number of bins** (typically 2 – 4). This observation is also confirmed by the results on real datasets (Sec. VI-C). We **develop a SIMD algorithm to perform histogram updates** for such cases (without using gather/scatter) as described below:

Consider the case of $[r_{min}^{n_1,n_2}\ ..\ r_{max}^{n_1,n_2}]$ overlapping with *two* bins, and denote the bin boundaries as r_l, r_m, and r_h. Instead of an explicit histogram, we maintain $\mathcal{K}$ counters in a SIMD register, one for each lane. We denote this register as `reg_count`. In addition, we maintain an auxiliary register (`reg_aux`) that contains the value r_m^2) splatted across the $\mathcal{K}$ lanes. After $\mathcal{K}$ distance (square) computations, we compare the results with `reg_aux` and increment the lanes of `reg_count` that passed the comparison test (masked add). After all $|n_1'| \times |n_2|$ comparisons, we obtain the total number of pairs that passed the test by summing up the $\mathcal{K}$ lanes in `reg_count` (denoted as $\texttt{count}_{<m}$). The number of comparisons that did not pass the comparison test ($|n_1'| \times |n_2|$ - $\texttt{count}_{<m}$) equals the value of the other bin. A similar method is used for cases with *three* (and *four*) histogram bins, by maintaining two (and three) counter and auxiliary registers.

Note that the above method outperforms scalar histogram updates up to a certain number of bins. For our platform, if the number of bins is greater than *six*, we resort to histogram updates in scalar rather than SIMD. However, such cases constitute a very small fraction of the overall pairs. We achieve **near-linear SIMD scaling (∼7.6–7.9×)**, as described in Sec. VI-B.

B. Efficient Parallel Kd-Tree Construction

In addition to the usual optimization metrics for kd-trees (such as bounding the number of particles per node), we also *bound the maximum distance between any two points within a tree-node*. Furthermore kd-tree construction is not scalable for a large number of nodes [36]. Instead, we divide the kd-tree construction into following two steps to facilitate parallelization.

1. Uniform Subdivision: Let r_{thresh} denote a user-defined threshold of the maximum distance between any two points in a tree-node. As a first step, we perform a uniform sub-division of space, and create a uniform grid, with resolution denoted by dim_X X dim_Y X dim_Z. Each particle is binned into one of the bins based on its spatial coordinates. To create enough parallel work, r_{thresh} is chosen in a way that the resultant number of tree-nodes is a small multiple of the total number of executing threads ($\sim\kappa\mathcal{P}\mathcal{M}$, where $\mathcal{P}$ denotes the number of cores per node, and $\mathcal{M}$ denotes the number of nodes). Thus, the number of nodes in each dimension is around $\sqrt[3]{\kappa\mathcal{P}\mathcal{M}}$. We employ a privatization approach where each node (and further each thread) computes a local binning, followed by a global reduction phase. We use κ = 100 for our runs. We present details of our multi-node implementation in Sec. V.

2. Non-uniform Subdivision: For each grid cell created above, we construct a kd-tree such that the number of points in each leaf-node is $\leq \mathcal{N}_{thresh}$, where $\mathcal{N}_{thresh}$ denotes the desired maximum number of particles within a node. We further parallelize the construction of kd-tree for all grid cells assigned to a node.

In practice, our two-step scheme for kd-tree construction scales well with overall number of cores ($\mathcal{P}\mathcal{M}$) and produces <1% extra tree-nodes as compared to a top-down method working on the complete dataset.

C. Load-Balanced TPCF Execution

We need to divide the tree-nodes (obtained at the end of uniform subdivision above) amongst the various executing threads so that each thread performs similar amount of execution to fully utilize the computational resources. However, since the time spent on each tree-node is not known beforehand, it is not possible to perform an apriori load-balanced distribution of work. We therefore divide the execution into two parts:
(1) A low-overhead initial **static division** of work.
(2) A **dynamic scheduling** of work such that all the computational resources are kept busy during the complete execution.

Our dynamic scheduling scheme is based on distributed Task Queueing [30] model, which is extended to multiple nodes. We use a randomized task stealing mechanism, similar to some recent works [34], [37]. Since our input datasets consist of large number of particles (one billion and greater), the task sizes are chosen in a way that latency of task transfer between nodes has minimal overhead on the run-time. We discuss intra-node and inter-node task queueing mechanism in detail in Sec. V. We in fact dedicate one of the $\mathcal{P}$ cores per node to task scheduling and bookkeeping. This core also takes care of all the MPI traffic, as described later.

V. Multi-Node Algorithm and Implementation

Given the input dataset $\mathcal{D}$, $n_{\mathcal{R}}$ catalogs of random particles ($\mathcal{R}_j$) are created and stored in individual files before the start of the execution of the TPCF algorithm. In order to implement the inter-node communication, we use the Message Passing Interface [38] (MPI) calls. We use the pthread library for threading within the node. We dedicate one thread (denoted as *Thread*$_{MPI}$) to handle all operations associated with MPI, including but not limited to system calls for copying data from user to internal buffers and back, etc. Furthermore, as mentioned in Sec. IV-C, we use a dedicated thread (*Thread*$_{TaskQ}$) to perform the dynamic task scheduling between various cores on a node, and amongst the nodes. Both these threads are mapped to the same core (Core$_0$). All other cores ($\mathcal{P}$-1) on each node perform computational tasks.

The TPCF execution involves computing $\mathcal{DD}(r)$ followed by $n_{\mathcal{R}}$ computations of $\mathcal{RR}_j(r)$ and $\mathcal{DR}_j(r)$ ($j \in [1 \,..\, n_{\mathcal{R}}]$). We assume $\mathcal{D}$ to be main memory resident throughout the execution, and iterate over $\mathcal{R}_j$'s by loading them one at a time, and computing the corresponding $\mathcal{DR}_j(r)$ and $\mathcal{RR}_j(r)$ and accumulating it to the final result. We also designate one node (say Node$_0$), where the final result (histogram $\omega(r)$) is accumulated and stored.

Step 1: Parallelizing Uniform Subdivision for $\mathcal{D}$ and $\mathcal{R}_j$'s.

Step 1.1: We divide the particles evenly between the $\mathcal{M}$ nodes, and each executing thread (($\mathcal{P}$-1) threads on each node) further divides the particles assigned to each node evenly among themselves. On each node, each executing thread (reads the data and) populates a local grid (with dim_X X dim_Y X dim_Z subdivisions). Once all threads are done, they compute a parallelized prefix sum to divide grid cells between threads, and cooperatively populate the grid.

Step 1.2: We now proceed to divide the grid cells between $\mathcal{M}$ nodes, such that each node owns similar number of particles. This is carried out in two phases. In the *first phase*, we evenly divide the cells between the nodes, and each node is sent the particles (and the count) that belong to the nodes assigned to it. Each node assimilates the particles received from other nodes to compute the particles (and count) for the grid cells assigned to it. At the end of *first phase*, each node has a count of the number of particles in each uniform grid cell. In the *second* phase, each node computes a prefix sum of the particle counts, and selects a contiguous set of grid cells so that the sum of number of particles in these cells is around $\frac{\mathcal{N}}{\mathcal{M}}$. The appropriate cells (particles and counts) are then transferred between the nodes. We use non-blocking `MPI_Isend` and `MPI_Irecv` calls.

At the end of Step 1, grid cells have been divided between the nodes so that each node owns similar number of particles. Furthermore, this mapping of grid cells to nodes is known to all the computational nodes.

Step 2: Parallel Kd-Tree Construction: For each assigned grid cell to the node, the ($\mathcal{P}$ − 1) threads cooperatively construct the kd-tree. For a cell with $\mathcal{N}_i$ particles, the top-down construction requires $\log(\mathcal{N}_i)$ phases, wherein for each phase, the splitting plane is computed, and particles divided between the two newly formed cells. We use a parallelized median computation [39] for this purpose. The division of particles between the two cells is also carried out in a SIMD- and thread-level parallel fashion. The termination criterion for the leaf nodes of the kd-tree was described in detail in Sec. IV-B. For small values of r_{max} (varying between 1–2%), the time spent in Step 2 varies between 2 – 5% of the total execution time, and given the large number of numbers and cores, it is important to efficiently parallelize this step. We achieve near-linear scaling, achieving >97% parallel efficiency for this step.

Step 3: Load-Balanced Parallel Tree Traversal and Histogram Update: We first discuss our load-balancing scheme. We start by first performing a *static sub-division of work* between nodes. Consider the computation of $\mathcal{DR}_j(r)$. Recall that Step 1 has already divided the uniform grid cells for $\mathcal{D}$ and $\mathcal{R}_j$ respectively between the nodes, such that the number of particles is almost evenly divided between nodes. Furthermore, the number of particles in each grid cell is also available locally on each node.

3.1 Static Subdivision of Work: For each assigned grid cell of $\mathcal{D}$ (say $\mathcal{G}^i_{\mathcal{D}}$), we first compute the neighboring grid cells of $\mathcal{R}_j$ within a distance of r_{max} of the cell. Let those grid cells be denoted by $\mathcal{G}^k_{\mathcal{R}_j}$. Let the total number of such neighboring cells be denoted by $\mathcal{N}_{ij}$. We approximate the amount of work as the sum of product of number of particles in $\mathcal{G}^i_{\mathcal{D}}$ and all neighbors ($\mathcal{G}^k_{\mathcal{R}_j}$). Hence,

$$\texttt{Work}_i = \sum_{k=1}^{\mathcal{N}_{ij}} |\mathcal{G}^i_{\mathcal{D}}| \cdot |\mathcal{G}^k_{\mathcal{R}_j}| \tag{2}$$

After each node computes $\texttt{Work}_i$ for all its assigned grid cells, it is broadcast to all other nodes (using `MPI_Allgather`). Each node (say $\texttt{node}_i$) then computes a prefix sum of the `Work`, and evenly divides the work between the nodes, and computes a subset of grid cells for which the node ($\texttt{node}_i$) will compute the TPCF for. This constitutes the static division of work between nodes. Note that the time taken to compute this static division of work is a very small fraction of the total run-time ($< 0.05\%$ for all our runs).

3.2 Communication of neighboring particles required for TPCF computation: Having computed the list of grid cells in $\mathcal{D}$ and $\mathcal{R}_j$ to compute the TPCF for, each node (say $\texttt{node}_i$) sends request for the grid cells that do not reside in its local memory to the respective 'owner' nodes of those grid cells (computed in step 1.2). The data is then transferred back so that the nodes can start with their TPCF computation.

3.3 Work division between threads on a node: As mentioned earlier, we dedicate a single thread for task management (*Thread*$_{TaskQ}$). Each grid cell assigned to a node is enqueued as a task by *Thread*$_{TaskQ}$. Each node maintains a `Signal`

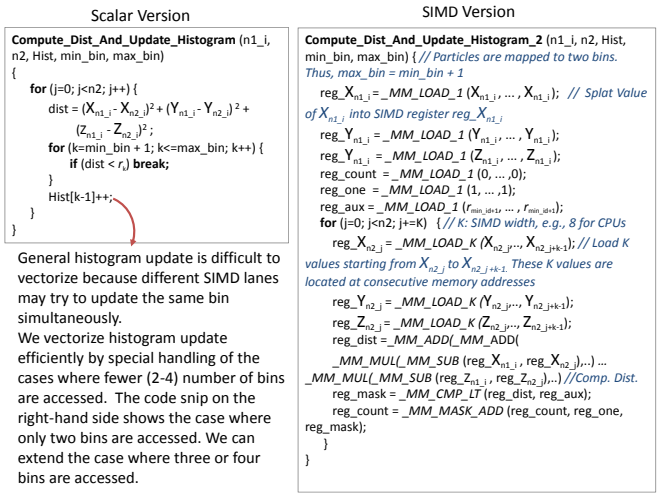

Scalar Version
SIMD Version
Compute_Dist_And_Update_Histogram (n1_i, n2, Hist, min_bin, max_bin)
{
for (j=0; j<n2; j++) {
dist = (Xn1_i - Xn2_j)2 + (Yn1_i - Yn2_j) 2 +
(Zn1_i - Zn2_j)2 ;
for (k=min_bin + 1; k<=max_bin; k++) {
if (dist < rk) break;
}
Hist[k-1]++;
}
}
General histogram update is difficult to vectorize because different SIMD lanes may try to update the same bin simultaneously.
We vectorize histogram update efficiently by special handling of the cases where fewer (2-4) number of bins are accessed. The code snip on the right-hand side shows the case where only two bins are accessed. We can extend the case where three or four bins are accessed.
Compute_Dist_And_Update_Histogram_2 (n1_i, n2, Hist, min_bin, max_bin) {// Particles are mapped to two bins. Thus, max_bin = min_bin + 1
reg_Xn1_i = _MM_LOAD_1 (Xn1_i , ... , Xn1_i); // Splat Value of Xn1_i into SIMD register reg_Xn1_i
reg_Yn1_i = _MM_LOAD_1 (Yn1_i , ... , Yn1_i);
reg_Yn1_i = _MM_LOAD_1 (Zn1_i , ... , Zn1_i);
reg_count = _MM_LOAD_1 (0, ... ,0);
reg_one = _MM_LOAD_1 (1, ... ,1);
reg_aux = _MM_LOAD_1 (rmin_id+1, ... , rmin_id+1);
for (j=0; j<n2; j+=K) {// K: SIMD width, e.g., 8 for CPUs
reg_Xn2_j = _MM_LOAD_K (Xn2_j... , Xn2_j+k-1); // Load K values starting from Xn2_j to Xn2_j+k-1. These K values are located at consecutive memory addresses
reg_Yn2_j = _MM_LOAD_K (Yn2_j... , Yn2_j+k-1);
reg_Zn2_j = _MM_LOAD_K (Zn2_j... , Zn2_j+k-1);
reg_dist = _MM_ADD(_MM_ADD(
_MM_MUL(_MM_SUB (reg_Xn1_i , reg_Xn2_j),..) ...
_MM_MUL(_MM_SUB (reg_Zn1_i , reg_Zn2_j),..) //Comp. Dist.
reg_mask = _MM_CMP_LT (reg_dist, reg_aux);
reg_count = _MM_MASK_ADD (reg_count, reg_one, reg_mask);
}
}

Fig. 3: Code snippet comparing the scalar version and SIMD version of `Compute_Dist_Update_Histogram`.

array with $(\mathcal{P}-1)$ 32-bit elements. We pre-define *three* 32-bit elements – `NEED_A_TASK`, `DONT_NEED_A_TASK` and `NO_TASKS_LEFT`, all greater than any of the task_id's.

All elements in `Signal` array are initialized to `NEED_A_TASK` – implying that the corresponding thread(s) needs a task to execute. The set of remaining tasks are initialized to the assigned set of tasks by the static subdivision of work. While there are remaining tasks, $Thread_{TaskQ}$ iterates over the `Signal` array, and in case any of the entries is `NEED_A_TASK`, replaces it with the head task_id of the remaining tasks and decrements the number of remaining tasks. The executing thread also polls the corresponding element in `Signal` array, and when it is replaced with a task_id, copies it to local space, and replaces the corresponding `Signal` element with `DONT_NEED_A_TASK` – implying that it does not need any task to execute, and executes the task (i.e. TPCF computation for all the particles in $\mathcal{G}_{\mathcal{D}}^{task\_id}$). We now discuss the SIMD alg. for Dual-Tree Traversal and Histogram Update.

Dual-Tree Traversal and Histogram Update: Fig. 3 (right) shows our SIMD friendly histogram update pseudo-code. Note that each of the grid cells have already been subdivided into individual kd-trees. The code is executed for all pairs of cells, formed by $\mathcal{G}_{\mathcal{D}}^{i}$, and $\mathcal{G}_{\mathcal{R}_j}^{k}$ (k $\in$ [1.. $\mathcal{N}_{ij}$]).

We traverse the pair of nodes and perform distance computation between node-node and particle-node pairs (`Compute_Min_Max_Dist_SIMD`). The other change from the baseline traversal is the SIMDfication of the key routines — namely, `Compute_Dist_Update_Histogram_2`, `Compute_Dist_Update_Histogram_3`, and so on, where the numeral value after the underscore is the number of bins that are overlapped for the particle-node interaction. As explained earlier in Sec. IV-A, these routines perform SIMD-friendly histogram updates for majority of the execution time, thereby increasing the utilization of the underlying computation resources. Note that `Compute_Dist_Update_Histogram_N` refers to the distance computation using SIMD, and histogram update using scalar ops.

Consider Fig. 3 (right). All instructions in the pseudo-code have corresponding AVX instructions, except the masked add (`_MM_MASK_ADD`). However, we exploit the fact that the preceding compare instruction (`_MM_CMP_LT`) in AVX (`_mm256_cmplt_ps`) returns the result in a register that can be subtracted from the original count (using two SSE ops – due to the unavailability of corresponding AVX instructions) to compute the final result.

As far as setting $\mathcal{N}_{thr}$ (the maximum number of particles in a kd-tree node) is concerned, setting it too small implies a large overhead for computing the range of distances between nodes, and setting it too large implies large overestimation of the range bounds, and hence disproportionate increase in computation. For current CPUs, setting $\mathcal{N}_{thr}$ to 200 – 250 strikes the right balance. For medium-to-large datasets, a large fraction of run-time ($> 98\%$) is spent in this step.

3.4 Dynamic Load-Balancing: On a node, while $Thread_{TaskQ}$ has remaining tasks, it distributes them among the executing threads. Once all the tasks are exhausted, the dynamic scheduling mechanism is used. The corresponding node (say $node_i$) selects a node ($node_j$) at random, and sends a request for tasks. In case $node_j$ has greater than a pre-defined threshold of tasks (denoted by `THRESH_TASK_STEALING`), it sends out the corresponding tasks to $node_i$. In case the remaining tasks are less than `THRESH_TASK_STEALING`, it sends out an empty range of tasks to $node_i$. In case $node_i$ doesn't receive any tasks, it chooses another random node and the process repeats. In case the process fails for `THRESH_ATTEMPT_TASK_STEALING` attempts, $Thread_{TaskQ}$ writes `NO_TASKS_LEFT`, implying that all tasks have been executed, at which point the threads go to a synchronization (barrier) point. $Thread_{MPI}$ sends out (and receives ack.) from the remaining nodes that it is done performing computation before going to the barrier. This ensures that the remaining nodes do not send any requests for tasks to this node. Once all the threads reach the barrier, the execution on the node terminates, and the resultant histogram is transferred to $Node_0$, the designated master for collecting and reducing the final TPCF results.

As far as the task management due to node-level task migration is concerned, we extend our intra-node task scheme by introducing an extra 32-bit element in the `Signal` array, which is set aside for node-level task migration, and polled by $Thread_{MPI}$. The rest of the logic follows in a straightforward fashion from our earlier description of per-node task management. Our protocol for polling for a maximum of `THRESH_ATTEMPT_TASK_STEALING` unsuccessful attempts ensures termination of the process. We empirically set `THRESH_ATTEMPT_TASK_STEALING` to around $\sqrt[3]{\mathcal{M}}$ to strike the right balance between sending out too many task stealing requests and achieving performance gains in practice. Also, note that

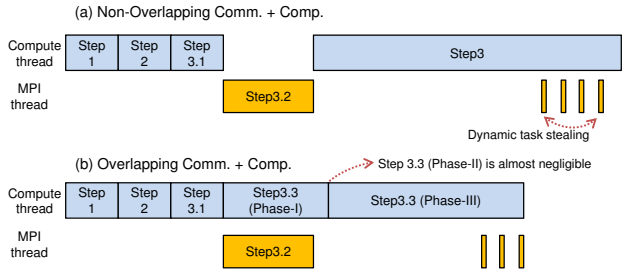


Fig. 4: Pictorial representation of the compute threads and the MPI thread for (top): Non overlapping of Steps 3.2 and 3.3, which keeps the compute thread idle while the data from relevant neighboring nodes is transferred to the node. (bottom): Overlapping communication and computation keeps the compute-threads busy (thereby increasing the compute efficiency) while the data is being transferred.

`THRESH_ATTEMPT_TASK_STEALING` reduces to the number of remaining nodes if their count is less than the initialized value of `THRESH_ATTEMPT_TASK_STEALING`.

Although many schemes for choosing the neighbor node are possible, random selection has been shown to be optimal [31], [34] for most scenarios. Finally, although the discussion in Step 3 has focused on computation of $\mathcal{DR}_j$, it is also applicable to $\mathcal{RR}_j$ and $\mathcal{DD}$ computation.

A. Performance Optimization: Overlapping Computation with Communication

In Step 3.2, we transfer all the neighboring grid cells required by a node before commencing with the TPCF computation on that node. Fig. 4 (top) shows the corresponding time-line. However, note that the node already has partial data to start the TPCF computation while the neighbor data is being transferred. Hence, we propose a 'hybrid scheme', overlapping computation and communication as described below. We divide the computation (Step 3.3) into *three* phases:

Phase-I: Perform TPCF computation using the available grid cells, and mark the missing cells (which are being transferred while the computation is taking place). No inter-node task migration is allowed during Phase-I.

Phase-II: Wait for neighbor data transfers to finish. For realistic input parameters, and due to the O($\mathcal{N}\sqrt{\mathcal{N}}$) order computational complexity, the data transfers are expected to take less time than the execution of Phase-I for current node-to-node transfer bandwidths. Hence, negligible (if any) time is spent in this phase for any of our runs.

Phase-III: Proceed with the remaining TPCF computation using the grid cells missing in Phase-I. Steps 3.3 and 3.4 are executed as explained. Inter-Node task migration is allowed, thereby exploiting the benefits of our node-level load-balancing scheme, while hiding the communication latency by executing the computation/comm. phases simultaneously.

Fig. 4 (bottom) shows the time-line with our 'hybrid approach'. As far as the speedups are concerned, even for large r_{max} sizes, we achieved a speedup of 1.05×, and the benefit increases for smaller r_{max} (∼1.15× for r_{max}=2.5%) due to the reduced amount of computation time spent per particle exposing the transfer latency. Our 'hybrid' scheme helps reduce the effect of this latency to **improve the achieved parallel node- and thread-level scaling**.

VI. Results

Platform Specification: We evaluate the performance of our novel TPCF implementation on two different Intel Sandy-bridge clusters – the Intel Endeavor cluster with 320 nodes and the Zin supercomputer at Lawrence Livermore National Laboratory, using 1600 nodes. Each node is an Intel Xeon E5-2670 with 16-cores (8-cores per socket, 2-way SMT) at 2.6 GHz. Each core has 256-bit SIMD (AVX) units that can perform multiplication and addition on *eight* 32-bit operands simultaneously, for a total of 665.6 GLOPS/node.

The 320-node Intel Endeavor cluster is connected through QDR InfiniBand 4X with the peak bandwidth 4 GB/sec and a two-level 14-ary fat-tree topology. The Zin system at Lawrence Livermore National Laboratory consists of 18 scalable units (SU). Each SU has 154 compute nodes for a total of 2772 compute nodes and 88.7TB of memory. Only 1600 nodes were used for this study, with a resultant of 1600 * 665.6 GLOPS∼1.06 Petaflops. We use the Intel ComposerXE 2011 compiler and the pthreads library for threading within a node. MPI [38] is used for inter-node communication.

Dataset sizes and other parameters: We vary the dataset size ($\mathcal{N}$) from 6.3 X 10^7 particles to 1.7 X 10^9 particles. The data was generated with a Particle-Particle-Particle Mesh N-body code. The particles represent dark matter that interact gravitationally and evolve into large scale structures. The data cubes for various redshifts are generated. The data used is for redshift zero. The random datasets ($\mathcal{R}_j$) are uniformly generated at random. Unless otherwise stated, r_{min} = 0.1% of $\mathcal{D}im$, and $\mathcal{H}$ = 10 bins. A total of $n_{\mathcal{R}}$ = 100 random datasets are pre-generated for each dataset (similar to other recent work [19], [20]). All measurements start the timings once the datasets have been loaded into main memory. All results are for single-precision floating point computation, which suffices for correlation computation on astronomical datasets.

Sec. VI-A reports the node-level scaling of our implementation. Sec. VI-B reports the speedup breakdown due to various optimization's and SIMD-friendly implementation. Sec. VI-C reports the run-times for varying dataset sizes, r_{max} and number of bins. Overall performance efficiency is derived in Sec. VI-D. Finally, we analyze the cost performance benefits of our scheme in Sec. VI-E.

A. Node-Level Scaling

Fig. 5 plots the node-level scaling for $\mathcal{N}$ = 1.73 Billion particles. Note that our node-level parallelization is based on an implementation that is already highly optimized (including SIMD) and parallelized within a node. On 1600-nodes we achieve 1550× parallel scaling using our static + dynamic load-balanced implementation coupled with the computation/communication overlapping scheme discussed in Sec. V. This implies a **node-level scaling efficiency of 96.8%** on our PetaFlop cluster. The resultant run-time was 19,118 seconds (5.3 hours), as opposed to a run-time of around 354

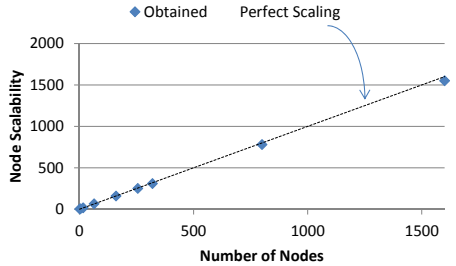


Fig. 5: **Node-Scalability of our algorithm. $\mathcal{N}$ = 1.73 X 10^9 particles, and r_{max} = 10%. Our static + dynamic load-balanced scheme coupled with computation/communication overlap achieves a parallel scaling of 1550× on 1600-nodes, an efficiency of 96.8%. We also draw the line of perfect scaling to show the scaling efficiency.**

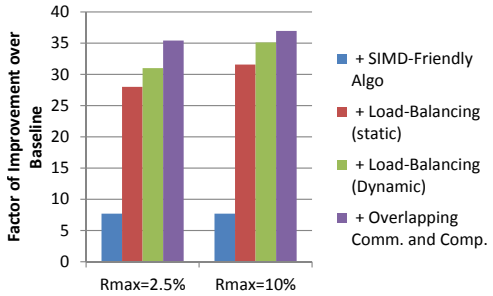


Fig. 6: **Relative benefit of our algorithmic optimization's for two different use-cases, r_{max} = 2.5% and 10% respectively (both with $\mathcal{N}$ = 1.73 X 10^9) as compared to a scalar uniform sub-division of work. $\mathcal{M}$ = 256 nodes for both cases. We achieve a SIMD scaling of 7.7× (max 8×) for both cases. Our static + dynamic load-balancing scheme achieves a further 4.0× (r_{max} = 2.5%) and 4.5× (r_{max} = 10%). Overlapping computation and communication provided a bigger boost for r_{max} = 2.5% (due to smaller run-time) to achieve an overall speedup of 35.4× (r_{max} = 2.5%) and 36.6× (r_{max} = 10%).**

days on a single-node [3]. Also, we obtained very small variance (∼0.15%) in run-times of $\mathcal{RR}_j(r)$ (for $j \in [1\ ..\ n_{\mathcal{R}}]$). The $\mathcal{RR}_j(r)$ computation scaled 1580×, while $\mathcal{DR}_j(r)$ scaled 1538×(∼0.47% variance).

B. Benefit of Optimizations

Fig. 6 shows a breakdown of the relative speedups obtained using various optimizations. Our base-line code is a scalar code parallelized using uniform division of cells between various nodes and threads — $\mathcal{N}$ = 1.73 X 10^9, $\mathcal{M}$=256 and r_{max} = 2.5% for the bars on left and 10% for the other set of bars.

The first bar shows the speedup obtained using our SIMD algorithm (Sec. V). Without using our approach, we noticed a SIMD scaling of less than 1.02× since only a small part of the distance computation was SIMDfied, while the histogram computation (which accounts for majority of the run-time) was still performed in scalar. Our algorithm enabled SIMD computation of histogram updates and improved the scaling by 7.7×, to obtain a resultant **data-level (SIMD) scaling efficiency of 96%**.

Consider the right set of bars (r_{max} = 10%) for this discussion. Using a uniform subdivision of grid cells between nodes, we achieved a scaling of 53× parallel scaling on 256-nodes. Using our formulation for static division of work dramatically improved our scaling by 4.1× to achieve 217× scaling. Adding the dynamic scaling improved the scaling by another 11% to 242×. Finally, overlapping computation/communication (instead of Step 3.2 in Sec. V) improved the final scaling to 254× on 256-nodes.

To summarize, *our overall run-time performance (and hence energy efficiency) improved by 35–37× using the algorithmic techniques developed in this paper.*

C. Effect of Varying Dataset sizes ($\mathcal{N}$) and r_{max}

Fig. 7 shows the relative increase in run-times for $\mathcal{N}$ = 1.73 X 10^9 and $\mathcal{M}$ = 256. Our run-time increases approximately as $O(\mathcal{N}\sqrt{\mathcal{N}})$ (similar to run-time increases reported by previous correlation computation papers). For comparison, we also plot $O(\mathcal{N}^2)$. In Fig. 8, we plot the relative increase in run-times with increasing r_{max}. The run-time varies (empirically) as $(r_{max})^{1.7}$ (also plotted for comparison purposes). Note that the run-time does not vary as $(r_{max})^3$ (the number of interactions) due to the existence of cases where the particles within pairs of tree-nodes fall within the same bin, and hence no actual computation is done (except incrementing the histogram bin by product of number of particles in the two tree-nodes). We also varied the number of histogram bins ($\mathcal{H}$). Increasing the number of bins from 10 to 20 increased the run-time by 1.3×, while increasing $\mathcal{H}$ to 30 increased the run-time by further 1.1×. The increase in run-time is due to the relative decrease in the number of tree nodes pairs whose histogram computation falls in the same bin.

[3]Run-times for ≤ 256 nodes were obtained by simultaneously using all the computational nodes in groups of nodes required for the specific data point. Thus each of the datapoints was obtained in a few hours as opposed to days.

D. Overall Performance Efficiency

We achieved a parallel scaling of 14.9× on each node, implying a (**thread-level scaling efficiency of 93.1%**). Note that the maximum scaling possible is 15×(=($\mathcal{P}$-1)), since we dedicate a core to manage MPI and task management for load-balancing. Combining with the node-level scaling of 1550× (Sec. VI-A) and SIMD-scaling of 7.7×(Sec. VI-B), we achieve a total performance scaling of 177,832× (max. 204,800× = (8)·(16)·(1600)×). Thus our algorithmic optimizations achieve an **overall performance efficiency of 86.8%**.

E. Cost Performance Analysis

The most relevant previous work is by Dolence and Brunner [1], which uses Abe, a NCSA Intel-64 cluster with peak compute of 89.47 TFLOPS (in 2007). In comparison, we use our 320-node cluster system with a peak of 212.9 TFLOPS (in 2012). Although difficult to perform apples-to-apples comparison, we conservatively achieve 11.1× performance gain (due to our SIMD speedup of 7.7×, and 1.43× higher core-scaling). Note that they give performance numbers on a much lower core count than the total cores available, but we scale their expected performance linearly for ease of comparison.

As far as the price is concerned, in absence of any pricing detail on their system, we use the following qualitative

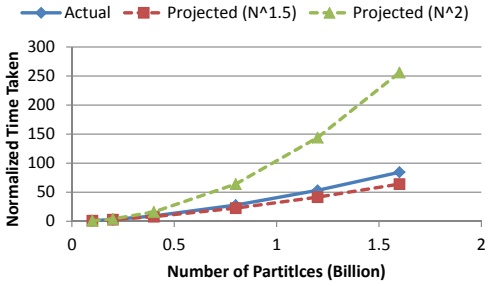


Fig. 7: **Relative increase in run-time with increasing number of particles. Previous works [1] have reported an approximate order-complexity of $\mathcal{N}^{1.5}$. The run-time scales close to the theoretical limit case of $\mathcal{N}^{1.5}$, which demonstrates we have reached flop-limited performance. We also plot the relative run-time without the use of dual-tree acceleration data-structures ($\sim\mathcal{N}^2$).**

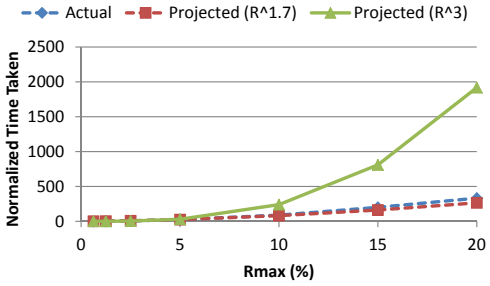


Fig. 8: **Relative increase in run-time with increasing r_{max} from 0.63% to 20%. Our run-time increases proportional to ${r_{max}}^{1.7}$, as opposed to ${r_{max}}^3$ (without any optimization's).**

argument. Moore's law dictates that the performance approximately doubles every 18 months for a similar cost budget. Since that system was deployed more than 4.5 years back, we should be able to achieve 715.76 peak TFLOPS for the same cost budget. However, our 320-node system as compared to their 1000-node system only provide for a peak of 212.9 TFLOPS, and is expected to be priced 3.3× lower. Another way to compare the price ratios is based on the argument of maintaining similar single node prices across different generations. The Abe system has 1000 nodes, while our system has 320 nodes – a price reduction of 3.12× for our system. Hence, we achieve a cost performance benefit in the range of 11.1– 36.63×, but report only 11.1× better flops/$ for a fair comparison.

VII. Related Work and Performance Comparison

Lot of recent research has focused on accelerating correlation functions on modern computing platforms. Kindratenko et al. [20] accelerate correlation computation using the O($\mathcal{N}^2$) variant on FPGAs and report >90× speedups over their unoptimized and unparallelized CPU implementation. However, our optimized algorithm on a single node is 80× faster than the FPGA performance numbers on a similar sized dataset (97.1K particles). With respect to multi-node performance, Roeh et al. [19] also parallelize correlation computation on a cluster of GPUs. They also implement the O($\mathcal{N}^2$) variant which performs orders of magnitude more work than a dual-tree based implementation on a single-node itself.

As far as load balancing of tree-traversal based applications across very large number of nodes is concerned, Dinan et al. [34] presented a randomized work stealing mechanism and achieved an efficiency of 86% on a cluster with 8K cores on a tree-search application. Recently Mei et al. [33] used the CHARM++ run-time for a 100-million atom NAMD simulation, and achieved a parallel efficiency of 93% on 224K cores (vs 6.7K cores). In comparison, our TPCF computation achieves 90% parallel efficiency, and an additional 96% SIMD efficiency. In the current system, we incur a $\frac{1}{16}$ (= 6.25%) performance loss since one core on each node is dedicated to MPI and task management. With increasing number of cores per node [8], we believe our efficiency would go further up.

The closest related work to our system is by Dolence and Brunner [1], who present a cluster-level parallelization of dual-tree based TPCF algorithm and reported speedups of about 700× using 1024 cores (128 dual-socket quad-core Xeons) (68% efficiency). In comparison, we achieve 23,095× speedup using 25,600 cores (90.2% efficiency), 1.32× higher efficiency on 25× larger core count using a combination of static + dynamic load-balancing scheme coupled with computation/communication overlap (Sec. VI-A). Additionally we also efficiently exploit SIMD, which their algorithm cannot.

The 2010 Gordon Bell prize winner, Hamada et al. [40] presented another astrophysical application – Barnes and Hut [41] $\mathcal{N}$-body simulation with 3.2 Billion particles on a 144-node GPU cluster. The run-time complexity of TPCF algorithm (O($\mathcal{N}\sqrt{\mathcal{N}}$)) is much higher than their run-time complexity (O($\mathcal{N}\log\mathcal{N}$)), and in fact, increases dramatically with increase in number of particles, and challenges the limits of exa-scale computing and beyond.

VIII. Conclusions

We presented a SIMD-, thread- and node-level parallel friendly algorithm for scaling TPCF computation on massive datasets, which brings down the execution time to a few hours for current-sized datasets with more than billion particles. We achieve 35–37× efficiency improvement over state-of-the-art techniques and have the potential for bringing an unprecedented level of interactivity for researchers in this field.

Acknowledgment

We would like to thank Dr. Ilian T. Iliev from University of Sussex for the input dataset. Hemant Shukla would like to acknowledge the ICCS project ISAAC funded through the NSF grant award #0961044 under PI Dr. Horst Simon. We also thank Scott Futral and Robin Goldstone for running experiments on the Zin supercomputer at Lawrence Livermore National Laboratory, Scott Mcmillan for his help with the initial experiments and Prabhat for his insightful discussions on improving efficiency.

REFERENCES

[1] J. Dolence and R. J. Brunner, "Fast Two-Point Correlations of Extremely Large Data Sets," in *9th LCI International Conference on High-Performance Clustered Computing*, 2008.

[2] "Accelerating Cosmological Data Analysis with FPGAs," 2009, http://www.ncsa.illinois.edu/ kindr/projects/hpca/files/ fccm09_presentation_v2.pdf.

[3] A. Ptak *et al.*, "Galaxy Evolution with LSST," *American Astronomical Society*, vol. 43, no. 16, pp. 217–252, 2011.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: a fast array of wimpy nodes," in *SOSP*, 2009, pp. 1–14.

[5] J. G. Koomey, "Worldwide electricity used in data centers," *Environmental Research Letters*, vol. 3, no. 3, p. 034008, 2008. [Online]. Available: http://stacks.iop.org/1748-9326/3/i=3/a=034008

[6] "Intel Advanced Vector Extensions Programming Reference," 2008, http://softwarecommunity.intel.com/isn/downloads/intelavx/Intel-AVX-Programming-Reference-31943302.pdf.

[7] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: A Many-Core x86 Architecture for Visual Computing," *Proceedings of SIGGRAPH*, vol. 27, no. 3, 2008.

[8] K. B. Skaugen, "HPC Technology — Scale-Up and Scale-Out," http://lecture2go.uni-hamburg.de/konferenzen/-/k/10940, 2010, keynote at SC10.

[9] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen, "Atomic vector operations on chip multiprocessors," in *ISCA*, 2008, pp. 441–452.

[10] J. Weiland, N. Odegard, R. Hill, E. Wollack, G. Hinshaw *et al.*, "Seven-Year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Planets and Celestial Calibration Sources," *Astrophys.J.Suppl.*, vol. 192, p. 19, 2011.

[11] P. J. E. Peebles and B. Ratra, "The cosmological constant and dark energy," *Rev. Mod. Phys.*, vol. 75, pp. 559–606, Apr 2003. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.75.559

[12] "The Nobel Prize in Phsics, 2011," 2011, http://www.nobelprize.org/nobel_prizes/physics/laureates/2011/index.html.

[13] "The Square Kilometer Array," 2011, http://www.skatelescope.org/wp-content/uploads/2011/03/SKA-Brochure_web.pdf.

[14] "Large Synoptic Survey Telescope PetaScale Data R&D Challenges," 2012, http://www.lsst.org/lsst/science/petascale.

[15] P. C. Broekema, A.-J. Boonstra *et al.*, "DOME: Towards the ASTRON AND IBM Center for Exascale Technology," IBM, Tech. Rep., 2012. [Online]. Available: http://domino.research.ibm.com/library/CYBERDIG.NSF/ papers/D30B773CBFF6EB39852579EB0051279C/$File/rz3822.pdf

[16] R. P. Norris, "Data challenges for next-generation radio telescopes," in *Proceedings of the 2010 Sixth IEEE International Conference on e-Science Workshops*, ser. E-SCIENCEW '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 21–24. [Online]. Available: http://dx.doi.org/10.1109/eScienceW.2010.13

[17] P. J. E. Peebles, *Large-Scale Structure of the Universe*. Princeton University Press, 1980.

[18] S. D. Landy and A. S. Szalay, "Bias and variance of angular correlation functions," *Astrophysical Journal*, vol. 412, no. 1, pp. 64–71, 1993.

[19] D. W. Roeh, V. V. Kindratenko, and R. J. Brunner, "Accelerating Cosmological Data Analysis with Graphics Processors," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2009, pp. 1–8.

[20] V. V. Kindratenko, A. D. Myers, and R. J. Brunner, "Implementation of the two-point angular correlation function on a high-performance reconfigurable computer," *Scientific Programming*, vol. 17, no. 3, pp. 247–259, 2009.

[21] J. W. Tukey, "Bias and confidence in not-quite large samples," *The Annals of Mathematical Statistics*, vol. 29, no. 2, p. 614, 1958.

[22] A. G. Gray and A. W. Moore, "'N-Body' Problems in Statistical Learning," in *Advances in Neural Information Processing Systems (NIPS)*, 2000, pp. 521–527.

[23] A. W. Moore, A. J. Connolly, C. Genovese, A. Gray, L. Grone, N. Kanidoris II, R. Nichol, J. Schneider, A. Szalay, I. Szapudi, and L. Wasserman, "Fast algorithms and efficient statistics: N-point correlation functions," in *Mining the Sky*, ser. ESO Astrophysics Symposia, A. Banday, S. Zaroubi, and M. Bartelmann, Eds., 2001, vol. 20, pp. 71–82.

[24] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, pp. 509–517, 1975.

[25] NVIDIA, "Fermi Architecture White Paper," 2009. [Online]. Available: $http$: $//www.nvidia.com/content/PDF/fermi\_white\_papers/ NVIDIA\_Fermi\_Compute\_Architecture\_Whitepaper.pdf$

[26] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey, "Sort vs. hash revisited: Fast join implementation on modern multi-core cpus," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, 2009.

[27] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: a case for bandwidth oblivious simd sort," in *SIGMOD Conference*, 2010, pp. 351–362.

[28] T. P. chun Chen, D. Budnikov, C. J. Hughes, and Y.-K. Chen, "Computer vision on multi-core processors: Articulated body tracking," in *ICME*, 2007, pp. 1862–1865.

[29] C. J. Hughes, R. Grzeszczuk, E. Sifakis, D. Kim, S. Kumar, A. P. Selle, J. Chhugani, M. Holliman, and Y.-K. Chen, "Physical simulation for animation and visual effects: parallelization and characterization for chip multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 220–231. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250690

[30] E. Mohr, D. A. Kranz, and R. H. Halstead, "Lazy task creation: a technique for increasing the granularity of parallel programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, pp. 185–197, 1991.

[31] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[32] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 162–173. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250683

[33] C. Mei, Y. Sun, G. Zheng, E. J. Bohm, L. V. Kalé, J. C.Phillips, and C. Harrison, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[34] J. Dinan, D. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the 2009 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 1–9.

[35] M. Deisher, M. Smelyanskiy, B. Nickerson, V. W. Lee, M. Chuvelev, and P. Dubey, "Designing and dynamically load balancing hybrid lu for multi/many-core," *Computer Science - R&D*, vol. 26, no. 3-4, pp. 211–220, 2011.

[36] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. B. Jr., S. V. Adve, and J. C. Hart, "Parallel sah k-d tree construction," in *High Performance Graphics*, 2010, pp. 77–86.

[37] A. Becker, G. Zheng, and L. Kale, "Distributed Memory Load Balancing," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer Verlag, 2011.

[38] "The Message Passing Interface (MPI) standard," http://www.mcs.anl.gov/research/projects/mpi/.

[39] D. A. Bader and J. JáJá, "Practical parallel algorithms for dynamic data redistribution, median finding, and selection," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 292–301. [Online]. Available: http://portal.acm.org/citation.cfm?id=645606.661174

[40] T. Hamada and K. Nitadori, "190 tflops astrophysical n-body simulation on a cluster of gpus," in *SC*, 2010, pp. 1–9.

[41] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, 1986.